ROVM Tutorial 문서

정 원교

2006.2.19

지구 남쪽에서

이메일 : weongyo@gmail.org

요 약

이 문서는 ROVM 을 처음 접하는 분들을 위한 문서로써 ROVM 을 어떻게 사용하는지와 ROVM 을 이용하여 어떤 일들을 할 수 있는지를 하나씩 쉽게 풀어 놓은 설명서입니다.

차 례

제 1 절 인사말

ROVM 세계에 오신 것을 환영합니다. (사실 이런 말로 시작해 보고 싶었습니다.)

ROVM 은 Remote Object Virtual Machine 의 약자로써 한글로 '원격 객체 가상 머신'으로 불릴 수 있습니다. '원격 객체 가상 머신'이란 말이 그렇게 어렵지는 않지만, 좀 더 쉽게 풀어 쓴다면 원격 (현재 자신의 컴퓨터가 아닌) 에 있는 객체 (Library 혹은 모듈, Class, Method, Variable) 를 이용하여 프로그램을 작성하는 것을 말합니다.

COM, CORBA, Java RMI 등등 이미 이와 비슷한 것이 존재한다는 것은 알지만 **한번 만들고 싶었습니다.** (ㅡ.ㅡ;)

이 문서는 이미 사용자가 ROVM Server 와 ROVM Client 를 설치하였다는 가정하에 작성이 되었습니다. 사용할 ROVM Server 가 없다면 최소한 다른 사람의 ROVM Server 의 IP (혹은 호스트이름) 와 Port 번호를 알고 있다고 가정합니다.

이 문서는 현재 (2006 년 2 월 19 일 기준으로) 몇 페이지 밖에 되지 않으며 작동되는 opcode 또한 고작 20 개 밖에 존재하지 않습니다. 이제 시간이 흐름에 따라 많은 opcode 들이 구현이 될 것이고 많은 operation 들을 사용자가 스스로 구현할 수 있을 때가 올 것입니다. 또한 그에 발맞추어 이 tutorial 또한 몇 백 페이지의 문서가 되어 있을 것입니다. 기대해 주세요

제 2 절 기본적인 작동 방식

ROVM 의 통신 방식에는 두가지가 존재합니다. 하나는 지금 이 글에서 언급하게 될 ROVM Server ↔ ROVM Client 통신 방법이고 나머지는 아직 구체화되어 있지는 않지만 ROVM Server ↔ ROVM Server 통신 방법입니다.

ROVM Server ↔ ROVM Client 통신 방법의 경우, 사용자는 ROVM Client 를 통해서 ROVM 에서 실행할 프로그램을 작성하고 Server 측에 보내게 되는 그것을 서버는 받아 실행되게 됩니다.

ROVM Server ↔ ROVM Server 통신 방법의 경우, 실행 중인 프로그램 (이미 시스템에 설치되어 있거나, 사용자가 보낸 ROVM opcode file format) 에서 필요한 Class Object (기타 등등) 가 다른 ROVM Server 에 존재할 경우, 그에 대한 결과값을 동적으로 요청하여 얻어오는 과정을 말합니다.

제 3 절 2 + 4 해보기

첫번째 예제로써 '2 + 4'를 하는 간단한 것을 해보겠습니다. 왜냐하면 2006 년 2 월 1 일 이 절을 작성하고 있는 시점에 ROVM Server 와 ROVM Client 로 할 수 있는 것은 `<char number> + <char number>` 가 전부이기 때문입니다.

3.1 Client 실행하기

이제 ROVM Client 를 실행시켜 봅시다. 아래와 같은 명령어를 통해서 쉽게 실행할 수 있을 것입니다.

```
C:\> python rovmclient.py
ROVM Client 0.0.1a
Type "help" for more information.
>>>
```

3.2 티켓 얻어오기

ROVM Server 의 리소스를 사용하기 위해서는 '티켓'을 끊어야 합니다. 이 '티켓'은 웹 서버의 Session 과 비슷한 개념으로 생각하시면 됩니다. 즉 여러분에게 할당된 VM 이 실행하는데 필요한 모든 정보들을 가지고 있으며, 마지막으로 동작했던 상태들을 그대로 저장해 놓고 있습니다.

'Garbage Collector' 에 의해 더 이상 사용되지 않는 티켓으로 인식이 되거나 여러분이 '**REQEND**' 명령어를 이용하여 직접 티켓을 폐기할 경우에서야 이 '티켓'은 ROVM Server 상에서 소멸되게 되는 것입니다.

티켓은 '**REQ'** 명령어를 통해서 얻어 올수 있습니다.

```
>>> req e://192.168.58.129:8888
Opening Socket 192.168.58.129:8888
Ticket '3K$\xe9\xb9\xfd\xc7\xb1\x9dqo\x07\xd3E\xae__\x16C\xbf'
Closing Socket.
>>>
```

'REQ' 명령어 뒤에 붙은 형식은 e://<hostname>:<port> 이라는 것 잊지 마세요.

3.3 Opcode 입력하기

이제 Ticket 를 끊었고 오류가 없었다면 이제 ROVM Server 에 실행할 명령을 보낼 수 있는 권한을 가지게 되었습니다. 이제 간단한 명령을 실행시켜 봅시다.

우선 ROVM Client 의 'Opcode 모드'로 진입을 하여 간단한 char 숫자를 stack 에 push 하여 더하기를 해봅시다.

```
>>> opcode
Using ticket '3K$\xe9\xb9\xfd\xc7\xb1\x9dqo\x07\xd3E\xae__\x16C\xbf'
... cpush 2
... cpush 4
... iadd
...
>>>
```

'Opcode 모드'로 진입하기 위해서는 opcode 명령어를 입력해야 합니다. '...' 프롬프트가 화면상에 출력되면 여러분

은 ‘Opcode 모드’에 현재 와 있다는 뜻입니다.

‘**cpush**’ 는 char 타입인 숫자를 스택에 push 하는 것입니다. ‘**iadd**’ 는 stack 에서 2 slot 을 pop 한 후 그것을 더한 값을 push 하게 됩니다.

‘Opcode 모드’를 빠져나오기 위해서는 단순히 <ENTER> 키만 눌러주면 됩니다.

아직 저희가 입력한 명령어가 ROVM Server 에 전달되지 않았습니다. ‘**SEND**’ 명령을 해줘야 실제로 전달되게 됩니다. 우선 ‘**SEND**’ 명령을 하기 전에 ‘**PRINTOPCODE**’ 를 입력해 봅시다. 사용자가 입력한 opcode 가 바이너리 형태로 변환되어 있음을 확인할 수 있습니다. 이 바이너리가 실제로 ROVM Server 에 전달되는 것입니다.

```
>>> printopcode
#0 '\x10\x02'
#1 '\x10\x04'
#2 '`'
>>>
```

3.4 Opcode 전송하기

이제 ‘**SEND**’ 명령어를 입력하여 실제 VM 이 실행하도록 해봅시다.

```
>>> send
Opening Socket 192.168.58.129:8888
Return VOID
Closing Socket.
>>>
```

실행시킨 후의 결과값으로 VOID type 이 반환되었습니다. 이것은 우리가 입력한 opcode 에는 return 값에 영향을 미치는 요소가 없었기 때문입니다.

3.5 상태 확인하기

그럼 서버의 스택 상태를 확인해 봅시다. ‘**STACK**’ 명령어를 통해서 쉽게 확인할 수 있습니다. 내용을 봐서 저희가 입력한 명령어가 성공적으로 실행되어 스택에 잘 들어가 있음을 확인할 수 있습니다.

```
>>> stack
Opening Socket 192.168.58.129:8888
[DEBUG] [Mon Jan 30 10:42:36 2006] proc_rc.c(170): #0 INT 0x6
Closing Socket.
>>>
```

3.6 티켓 버리기

이제 사용자가 할당한 Ticket 을 모두 사용하였고 더 이상 사용하지 않을 경우라면 이제 티켓을 버리도록 합시다. 'REQEND' 명령을 통해서 해당 사항을 수행할 수 있습니다. 'OK' 메세지를 받았다면 이제 해당 티켓은 ROVM Server 에서 완전히 소멸되었으며 더 이상 해당 티켓으로는 어떠한 명령어도 실행시킬 수가 없습니다.

```
>>> reqend
Opening Socket 192.168.58.129:8888
<- OK
Closing Socket.
>>>
```

제 4 절 '티켓'의 개념 파악하기

이 절에서는 ROVM 에서 사용하는 '티켓 (이하 Ticket)'에 대한 개념에 대해서 설명을 하도록 하겠습니다.

Ticket 은 두가지 모습을 가지고 있습니다. 웹의 Session 과 비슷한 개념이라고 제가 앞에서 설명한 적이 있습니다.

- 사용자 측에서 바라보았을 때의 모습
- ROVM Server 내부에서 바라보았을 때의 모습

이 그것입니다.

사용자 측에서 바라 보았을 경우, 오직 Ticket 의 unique ID 만 보게 됩니다. 모든 정보는 이 unique ID 를 이용하여 가져오기, 실행하기, 쓰기 등등의 모든 operation 을 수행할 수 있습니다. 이 unique ID 를 '**Ticket ID**' 라고 부릅니다.

예를 들어 설명하면, 여러분이 영화를 보러갔을 때, 사게 되는 영화표랑 같은 모양입니다. 영화표는 여러분이 해당 영화를 볼 권리를 가지고 있음을 나타냅니다. 그것을 이용하여 영화를 볼 수 있고 입장을 할 수 있게 되지요. 만약 영화표가 없다면 해당 영화를 볼 권리를 가지지 못합니다. 하지만 영화표에는 권리만 있지 영화 전체 내용이 들어가 있지는 않습니다.

영화 전체 내용에 해당하는 것이 ROVM Server 에서는 **Ticket 구조체**입니다. 여러분이 Ticket ID 를 가지고 수행하는 결과물들이 모두 'Ticket 구조체'에 저장되게 되며, 나중에 이를 계속 이용하게 됩니다.

요약해서 말하면 'Ticket ID' 와 'Ticket 구조체'는 서로 한쌍으로 묶여 있으며, 'Ticket ID' 는 'Ticket 구조체'를 찾기 위한 unique key 라는 사실입니다.

실제 'Ticket 구조체'가 어떻게 동작하는지 살펴보도록 하겠습니다. 우선 ROVM Server (ROVM Server 가 192.168.58.129 의 8888 번 포트에 열려 있다고 가정하겠습니다.) 를 실행시키고 ROVM Client 를 실행시킵니다. 그리고 'Ticket' 을 얻어 옵니다.

```
C:\projects\rovmclient>python rovmclient.py
ROVM Client 0.0.1a
Type "help" for more information.
>>> req e://192.168.58.129:8888
Opening Socket 192.168.58.129:8888
Ticket '\xfc1\x0e\x0cA\xd2\xbaAy(d\xcbL\xf9\xa8X4\xce\x8e\x9a'
Closing Socket.
>>>
```

얻어온 Ticket ID 가 보입니다. 바이너리 형태이기 때문에 저렇게 보이는 것입니다. 이제 opcode 를 실행시켜 보겠습니다. ROVM Server 의 특징 중에 하나는 opcode 를 하나씩 실행할 수 있다는 점입니다. (물론 루프 형태일 경우는 힘듭니다만.)

```
>>> opcode
Using ticket '\xfc1\x0e\x0cA\xd2\xbaAy(d\xcbL\xf9\xa8X4\xce\x8e\x9a'
... cpush 65
...
>>> send
Opening Socket 192.168.58.129:8888
Return VOID
Closing Socket.
>>> stack
Opening Socket 192.168.58.129:8888
[DEBUG] [Tue Jan 31 04:04:27 2006] proc_rc.c(167): #0 CHAR 0x41
Closing Socket.
>>>
```

'CPUSH 65' 를 실행시키고, 스택 상태를 확인한 결과 스택에 잘 올라가 있음을 확인할 수 있습니다.

```
>>> opcode
Using ticket '\xfc1\x0e\x0cA\xd2\xbaAy(d\xcbL\xf9\xa8X4\xce\x8e\x9a'
... cpush 4
...
>>> send
Opening Socket 192.168.58.129:8888
Return VOID
Closing Socket.
>>> stack
Opening Socket 192.168.58.129:8888
[DEBUG] [Tue Jan 31 04:04:58 2006] proc_rc.c(167): #1 CHAR 0x4
[DEBUG] [Tue Jan 31 04:04:58 2006] proc_rc.c(167): #0 CHAR 0x41
Closing Socket.
>>>
```

다시 'CPUSH 4' 를 실행시키고, 스택 상태를 확인해 보았습니다.

```
>>> opcode
Using ticket '\xfc1\x0e\x0cA\xd2\xbaAy(d\xcbL\xf9\xa8X4\xce\x8e\x9a'
... iadd
...
>>> send
Opening Socket 192.168.58.129:8888
Return VOID
Closing Socket.
>>> stack
Opening Socket 192.168.58.129:8888
[DEBUG] [Tue Jan 31 04:05:33 2006] proc_rc.c(170): #0 INT 0x45
Closing Socket.
>>>
```

마지막으로 'IADD' 명령어를 실행하였습니다. 지금 말씀드리는 거지만, ROVM Client 와 ROVM Server 는 Connection 을 계속 유지하고 있지 않습니다. 위의 메세지 중 'Closing Socket' 이라고 되어 있는 부분은 Connection 을 끊었다는 의미입니다. 즉, 'REQ' 명령어 다음부터의 operation 은 모두 Ticket ID 를 통해서 자료 교환이 이루어지기 때문에, 연결을 유지할 필요가 없는 것입니다.

위의 예제에서 볼 수 있듯이, 'Ticket 구조체'에는 현재 Ticket ID 로 인해 발생하는 operation 이 저장되어 있음을 눈으로 확인할 수 있습니다.

```
>>> reqend
Opening Socket 192.168.58.129:8888
<- OK
Closing Socket.
>>>
```

마지막으로 'Ticket' 을 제거합니다. 이 명령어를 통해서 모든 'Ticket ID', 'Ticket 구조체'가 ROVM Server 에서 제거됩니다.

제 5 절 Class 다루기

이 절에서는 ROVM Server 에서 클래스를 어떻게 다룰지를 이야기하고자 합니다.

5.1 Object Reference 얻어오기

이 섹션에서는 ROVM Server v0.0.1b 에서 새롭게 추가된 'NEW' opcode 에 대한 사용법을 익혀 보도록 하겠습니다. 'NEW' opcode 는 ROVM 의 전체 opcode 들 가장 중요한 opcode 중 하나로써, 사용하고자 하는 Class 의 객체를 얻어오는 과정입니다.

C++ 이나 Java 등등 OOP (Object-Oriented Programming : 객체지향 프로그래밍) 에서 사용하는 '*this*' 혹은 '*self*' 에 해당하는 것이 바로 Object Reference 입니다.

이 Object Reference 는 Class 의 field 나 method 등등 클래스와 관련된 모든 정보들에 대한 접근을 위해서 필요합니다. 즉 Object Reference 포인터를 이용하면 해당 Class 의 모든 정보에 접근할 수 있습니다.

ROVM 의 'NEW' opcode 를 통해서 Object Reference 를 얻는 과정에 대해서 설명을 하도록 하겠습니다.

ROVM Client 를 실행시켜 Ticket 을 하나 얻습니다. (ROVM Server 가 192.168.58.129 의 4390 포트에 열려있다고 가정합니다.)

```
C:\projects\rovmclient>python rovmclient.py
ROVM Client 0.0.1a
Type "help" for more information.
>>> req e://192.168.58.129:4390
Opening Socket 192.168.58.129:4390
Ticket 'Q^\x90?3\xed>.\xa1#h\xf4\xf6\xca\xf5\xa7&\xf3\xe2\xb6'
Closing Socket.
```

그리고 'OPCODE' 명령어를 이용하여 'new' opcode 를 삽입합니다. 'new' opcode 의 형식은 "ROVM Client" 문서에 자세히 나와 있으니, 해당 문서를 참조하시기 바랍니다.

```
>>> opcode
Using ticket 'Q^\x90?3\xed>.\xa1#h\xf4\xf6\xca\xf5\xa7&\xf3\xe2\xb6'
... new e://192.168.58.129:4390/ABCDEF
...
>>> send
Opening Socket 192.168.58.129:4390
Return VOID
Closing Socket.
```

'SEND' 명령을 통해서 해당 명령을 실행시킨 결과, VOID 형식의 return 값이 반환되었음을 확인할 수 있습니다. 'new' opcode 는 반환값에 영향을 미치는 요소가 존재하지 않기 때문에, VOID 형식이 맞습니다. ROVM Client 측에서는 저 명령을 통해서 어떤 과정이 실행되었는지 눈으로 확인할 수는 없지만 그에 대한 결과값인 Object Reference 가 stack 에 잘 push 되었는지는 확인할 수 있습니다.

```
>>> stack
Opening Socket 192.168.58.129:4390
[DEBUG] [Fri Feb  3 02:45:56 2006] proc_rc.c(174): #0 OBJREF 0x808c8d8
Closing Socket.
>>>
```

'STACK' 명령어를 통해서 현재 Ticket 에 대한 스택 상태를 확인해 본 결과 제대로 'OBJREF' 타입의 Object Ref 가 올라가 있는 것을 확인할 수 있습니다.

앞으로 이 ObjRef 를 통해서 클래스의 Method 를 호출하거나 Field 값을 구하거나 클래스의 상속 개념이 생긴다면 해당 인터페이스로 접근할 수 있게 되는 것입니다.

```
>>> reqend
Opening Socket 192.168.58.129:4390
<- OK
Closing Socket.
>>>
```

마지막으로 연결을 마무리하고 Ticket 를 버립니다.

5.2 String Reference 만들기

이 절에서는 String Reference (이하 StringRef) 에 대해서 알아보도록 하겠습니다.

StringRef 는 단순하게 ROVM 에서 사용되는 "문자열" 이라고 생각하시면 됩니다. C 언어에서 `char *msg="안녕하세여?"` 와 같이 선언하는 것과 같다고 생각하시면 이해하기 쉬울 것입니다.

하지만 ROVM 상에서의 문자열은 “객체”이기 때문에, 문자열이라는 이름을 사용하지 않고 StringRef 라고 쓰는 것입니다.

Java, Python, Ruby 같은 스크립트 업어에서 사용해 보면 아시겠지만, 문자열에 함수가 선언되어 있는 모습을 보실 수 있을 것입니다. 그러한 것들이 ROVM 상에서도 가능합니다.

ROVM Server 에서는 ‘SPUSH’ 라는 opcode 가 존재하는데, 이는 C 언어 형태의 문자열을 StringRef 로 변환하여 스택상에 로드하는 명령어 입니다. 이를 이용하여 여러 문자열을 다루는 과정을 아래에서 설명하도록 하겠습니다.

```
C:\projects\rovmclient>python rovmclient.py
ROVM Client 0.0.1a
Type "help" for more information.
>>> req e://192.168.58.129:4390
Opening Socket 192.168.58.129:4390
Ticket 'Vc;d\x8f"\x91\xbc2\xcc\x051[\xe2C\x15E2\x13\x8e'
Closing Socket.
>>>
```

우선 티켓을 만들고 ‘SPUSH’ 명령을 사용해 봅시다.

```
>>> opcode
Using ticket 'Vc;d\x8f"\x91\xbc2\xcc\x051[\xe2C\x15E2\x13\x8e'
... spush "Hello ROVM Server"
...
>>> send
Opening Socket 192.168.58.129:4390
Return VOID
Closing Socket.
>>>
```

이제 스택을 확인해 봅시다. 아래와 같이 Stack 상에 StringRef 가 올라가 있는 것을 확인할 수 있습니다. 즉, 위에서 입력한 문자열이 ROVM 서버상에 로드되어 문자열 객체로 변환되었다는 이야기 입니다.

```
>>> stack
Opening Socket 192.168.58.129:4390
[DEBUG] [Fri Feb 10 01:31:50 2006] proc_rc.c(186): #0 STRINGREF 0x40236008
Closing Socket.
>>>
```

그럼 이 문자열을 이용하여 좀 더 재미있는 일들을 해보겠습니다. 현재 스택상에 올라가 있는 것은 StringRef (문자열 객체) 이기 때문에 이 객체 내부에 선언되어 있는 함수를 호출하여 그에 대한 결과값을 얻어 봅시다.

현재 문자열 객체 내부에는 split 라는 메쏘드가 선언되어 있습니다. 이것을 이용하여 “Hello ROVM Server” 을 배열로 나누어 봅시다. 위에서 계속 진행해 보겠습니다.

```
>>> opcode
Using ticket 'Vc;d\x8f"\x91\xbc2\xcc\x051[\xe2C\x15E2\x13\x8e'
... call split (T)[
...
>>> send
Opening Socket 192.168.58.129:4390
Return Array [(9, 'Hello'), (9, 'ROVM'), (9, 'Server')]
Closing Socket.
>>>
```

문자열 객체 내부에 선언되어 있는 split 메쏘드를 호출하였습니다. 그에 대한 결과값이 전송이 되었습니다. 3 개의 배열이 전송이 되었는데, Whitespace 를 기준으로 나누어진 모습이군요!

5.3 Method 호출하기

이 절에서는 ROVM Server v0.0.7a (버전 명명 규칙을 좀 바꿔 봤습니다. 7 의 의미는 opcode 가 7 개 존재한다는 의미이며, 뒤의 a 는 7 개 존재하는 VM 에서의 버전입니다.) 에서 새롭게 추가된 ‘CALL’ opcode 를 이용하여 Call 내의 method 를 호출하는 방법에 대해서 이야기 하도록 하겠습니다.

아래의 경우, ROVM Server 가 192.168.58.129 의 4390 포트에 열려 있다고 가정을 하였습니다.

```
C:\projects\rovmclient>python rovmclient.py
ROVM Client 0.0.1a
Type "help" for more information.
>>> req e://192.168.58.129:4390
Opening Socket 192.168.58.129:4390
Ticket '\xf6>t\xe7\xd7|\xd4\x99\x10\x8e\x10.\xab\xf3\xce\xfd5\xa1rT'
Closing Socket.
>>> opcode
Using ticket '\xf6>t\xe7\xd7|\xd4\x99\x10\x8e\x10.\xab\xf3\xce\xfd5\xa1rT'
... new e://192.168.58.129:4390/ABCDEF
... cpush 1
... cpush 2
...
>>> printopcode
#0 (29) '\xbb\x00\x0e\x00192.168.58.129&\x11\x07\x00/ABCDEF'
#1 (02) '\x10\x01'
#2 (02) '\x10\x02'
>>> send
Opening Socket 192.168.58.129:4390
Return VOID
Closing Socket.
>>> stack
Opening Socket 192.168.58.129:4390
[DEBUG] [Sat Feb  4 19:09:35 2006] proc_rc.c(168): #2 CHAR 0x2
[DEBUG] [Sat Feb  4 19:09:35 2006] proc_rc.c(168): #1 CHAR 0x1
[DEBUG] [Sat Feb  4 19:09:35 2006] proc_rc.c(174): #0 OBJREF 0x401e4008
Closing Socket.
```

```
>>> opcode
Using ticket '\xf6>t\xe7\xd7|\xd4\x99\x10\x8e\x10.\xab\xf3\xce\xfd5\xa1rT'
... call abc (TII)I
...
>>> printopcode
#0 (13) '\xb6\x03abc\x06\x00(TII)I'
>>> send
Opening Socket 192.168.58.129:4390
Return INT = 3
Closing Socket.
>>> stack
Opening Socket 192.168.58.129:4390
Closing Socket.
>>>
```

위의 과정을 보시면 아시겠지만, ABCDEF 클래스를 로드를 하여 ObjectRef 를 하나 생성하고, 'call' opcode 를 실행하기 전에 method 의 argument 가 되는 값들을 stack push 하였습니다. 그리고 함수를 호출한 결과가 나왔습니다.

이전에 ROVM Client 를 써보셨던 분들이라면 눈치채셨을지 모르겠지만, 'SEND' 명령어를 통해서 반환된 값이 이전과는 다른 모습임을 확인할 수 있을 것입니다.

기존의 모든 명령은 "Return VOID"가 왔었는데, 처음으로 "Return INT" 가 왔기 때문입니다. 이것은 실행시켰던 명령중에서 Return 값을 변환하는 것이 나왔다는 뜻입니다. 즉, 'CALL' opcode 에서 해당 값을 리턴한 것입니다.

참고로 위에서 실행한 abc 메쏘드의 구현은 아래와 같이 되어 있었습니다. 아래 문법의 이해는 "ROVM Compiler 사용 설명서" 를 참조하시기 바랍니다.

```
.class ABCDEF
    .def __init__ (T)V
        nop
    .defend

    .def abc (TII)I
        iload 1
        iload 2
        iadd
        ireturn
    .defend
.classend
```

위의 abc 메쏘드는 두번째 argument 와 세번째를 stack 상에 PUSH 한 후, 이를 더한 값을 반환하도록 되어 있습니다.

제 6 절 맺음말

Tutorial 을 끝까지 읽어 주셔서 감사합니다. 내용이 짧죠? 조금씩 Opcode 가 추가될 때마다 업데이트 하도록 하겠습니다.